INTL-0463-US
(P9817)

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: SHARING CLASSES BETWEEN PROGRAMS

INVENTOR: JOHN L. REID

Express Mail No. EL669040080US

Date: January 8, 2001

SHARING CLASSES BETWEEN PROGRAMS

Background

This invention relates generally to processor-based systems and particularly to the ability to share a class between two applications or programs.

A class, in object-oriented programming, refers to a group of smaller objects. Each object is defined by its data and the operations that can be performed on its data. These operations are the object's interface.

It is common for two software programs running on the same computer system to communicate and share data. Modern operating systems provide various low-level mechanisms to support data sharing between applications running on the same computer. Unfortunately, these mechanisms generally involve procedure-based access to shared memory. Since a large percentage of modern software projects are developed using object-oriented programming languages and object-oriented paradigms, these operating system supported mechanisms may be awkward and complex to implement, test and maintain when applied with object-oriented programming languages.

On Windows operating systems, one solution is the component object model or COM. While COM provides an object-oriented way to communicate and share data across

the boundaries of a program's address space, COM also involves substantial overhead. This overhead comes in the form of both a steep learning curve for developers and in the form of run-time overhead needed to install all the features COM publishes, whether or not the program actually uses those features. This has implications on the amount of resources a program uses, its performance, debug complexity, maintenance and modifiability.

Thus, there is a need for better ways to enable application programs to communicate and share data in an object-oriented fashion.

Brief Description of the Drawings

Figure 1 is a schematic depiction of one embodiment of the present invention;

Figure 2 is a more detailed schematic depiction of the embodiment shown in Figure 1; and

Figure 3 is a flow chart for software for implementing the embodiment shown in Figure 2.

Detailed Description

Referring to Figure 1, a pair of application programs 10a and 10b running on the same or different processor-based systems may communicate with one another through stub classes 12a and 12b. The stub classes 12a and 12b provide an interface between each application 10 and a shared class 12. The stub classes 12a and 12b hide the fact that the

class 12 actually being used is shared between two or more applications 10.

In some embodiments, the applications 10a and 10b may run on the same processor-based system. In another embodiment of the present invention, each application may run on a different processor-based system and those processor-based systems may be coupled together in a network.

The stub classes 12a and 12b then provide the glue that enables the applications 10 to share the same class 12 in a transparent or seamless fashion as seen by each application 10. The class method invocations may be invoked by either application 10 to instantiate a shared class object.

To share the class 12, the applications 10 may use a backing storage that comes from shared memory. Commonly, backing storage is allocated out of a program's default memory. As examples, backing storage may be allocated out of the program's global memory pool or out of stack memory. However, neither of these methods enable the backing storage for the class to be shared.

To provide shared memory, using the C++ programming language, the placement new operator may be utilized in one embodiment. The placement new operator allows one to specify the memory that is used as backing storage for any class objects that may be created:

```
Class* object = new(memory) Class;
```

Thus, the above code demonstrates one way to use the placement new operator to specify that the backing storage comes out of shared memory. Since the class object is to be shared, only one object may be instantiated and that object may then be shared. The mechanics of object execution can be encapsulated into a class that is used instantiate shareable objects. This encapsulation simplifies usage and can be used to guarantee that only a single instance is created. Referring to Figure 2, this class object is the share object 19. The share object 19 sets up the shared memory, for example, using services provided by the operating system. The share object 19 also instantiates the class to be shared out of shared memory 17. Since the share object 19 is independent of the type of class to share, this information is provided to the share object 19 at compile time as a parameterized type using C++ templates.

For example, the share object 19 may instantiate an arbitrary class to share:

```
Share<Class> share;
Class* object = share.instance();
```

Thus, each application program 10a and 10b may have an application specific memory 15a or 15b and may also utilize the shared memory 17. However, the shared memory 17 may include application specific portions 21a and 21b. Thus, object defined memory data that is program specific may be

duplicated in the portions 21a and 21b of shared memory 17 allocated to each application 10a or 10b.

A member is a variable or routine that is part of a class. A structure relates to the way data is organized in a storage. Using structure, one can refer to stored data locations without needing memory offsets. Instead, a symbolic reference to memory locations in the structure may be used. Thus, the structure may include a system in which data associated with certain variables is located at known locations or offsets within the storage.

In some object oriented programs, the structure may also include a function. When the function is called within the structure, the function may go out and access the needed data. Thus, member data may include the data stored in the structure as well as functions stored in the structure.

When a function within a structure is called, the appropriate data is recalled. The member data is commonly private to one application and, from outside the structure, member data may not be accessible by any application except the application responsible for that data. In effect, the structure encapsulates the data and gives it a well defined interface for getting and setting the data in the structure. At the same time, this organization decouples the actual layout of the memory from external view. The structure may correspond to a class and the functions that

are part of the structure or class may be called methods or functions of the class.

Thus, referring to Figure 2, the share object 19 creates the class object 12 which is shared by the applications 10a and 10b. The shared memory 17 is used by both applications 10a and 10b. Thus, each application 10a and 10b may have member data in the class object 12 and this data is specific to one application's address space. It would then normally be illegal for one application 10 to use that member data from another application's address space. In operating systems, when an application is started, the operating system assigns virtual address space to the application. An application may use a specific pointer to access the application's own address space. If another application attempts to use the pointer to access that data, the pointer is no longer valid. This is because each application has a different assigned virtual address space.

This creates a problem in having the two applications 10 share a class out of shared memory 17. While an object can be shared out of shared memory, the member data associated with the object may be specific to the address space of one of the application 10a or 10b. Therefore the other application 10a or 10b may be unable to access it.

This problem may be overcome by automatically duplicating member data in each address space 21 associated

with each application 10. Before a class object 12 is used, the share object 19 is created. Before each application 10 uses the share object 19, it calls an INIT method 25. The INIT method 25 creates the object INIT that makes sure that pointers to any member data specific to one application 10 is duplicated in the shared memory 12 associated with the other application 10. When the pointers and the associated data are duplicated, the INIT method 25 returns a pointer or handle to each application 10.

When an application 10 calls any other methods on the object 12, the application 10 uses that pointer or handle 27 to look up the correct application specific pointers. As a result, each application 10a and 10b has its own handle 27 to get to its own member data out of shared memory 17. All member data is accessible to both applications 10 because that member data (or at least pointers to it) are automatically duplicated.

It is desirable that each application 10 not be aware of specific member data in the class object 19 to be shared. Thus, the sharable class object 12 provides an Init method 25 that is invoked to perform this duplication. The shareable class object 12 implements a shareable interface 23 and thereby exposes methods of its own as well as those defined by the shareable interface 23, namely Init 25 and Uninit 29. Prior to using the object 12, the

applications A and B invoke the shareable interface 23 and Init method 25. Inside the method 25, the object duplicates any process specific data and returns a handle to the process or application 10 as indicated at 27.

The handle is then provided back to the object 12 in method invocations. The object 12 then uses the handle to resolve to the appropriate process specific member data:

```
void* handle = object -> Init();
object->method(handle, …);
```

In some embodiments of the present invention, a shared class may be utilized by two different applications using backing storage for the shared class in shared memory. However, all object defined member data which is process specific may be duplicated in the address space of all applicable processes in one embodiment.

Referring to Figure 3, the software 16, that may be stored on each processor-based system that shares classes, may begin by defining a share class that includes the object 19 of the type class that includes the object 12 as indicated in block 18 (share <class> share). Using the share class, an instance of the class to share is executed, as indicated in block 20 (class* object = share.instance ()). The shareable interface 23 of the class 12 is invoked to get an application specific handle ("context") 25, as indicated in block 20 (void* handle = object -> Init()). An application then uses the class object 12 as normal but specifies a handle in each method call to resolve its

specific context, as indicated in block 24, (object -> method (handle, …)). In some cases, this handle manipulation may be hidden in a stub or glue wrapper class.

While the present invention has been described with respect to a limited number of embodiments, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this present invention.

What is claimed is: